Section 04: Solutions

Section Problems

1. Big-$\mathcal{O}$ Red-Black Trees and BSTs

Write down a tight big-$\mathcal{O}$ for each of the following. Unless otherwise noted, give a bound in the worst case.

(a) Insert and find in a BST.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is "`log(n)`", but we're asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\mathcal{O}(n)$.

(b) Insert and find in a Red-Black tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively. Red-Black trees are guaranteed to be balanced, so the worst case is we need to traverse the height of the tree.)

(c) Finding the minimum value in a Red-Black tree containing n elements.

Solution:

$\mathcal{O}(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

(d) Finding the k-th largest item in a Red-Black tree containing n elements.

Solution:

With a standard Red-Black tree implementation, it would take $\mathcal{O}(n)$ time. If we're located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

(e) Listing elements of a Red-Black tree in sorted order

Solution:

$\mathcal{O}(n)$. We can just perform an in order traversal of the tree once, listing the contents of each node seen in the traversal.

2. Hashing

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$ (don't worry about resizing the internal array):

0, 4, 7, 1, 2, 3, 6, 11, 16

Solution:

To make the problem easier for ourselves, we first start by computing the hash values and initial indices:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

(b) Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$.

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

Solution:

Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelihood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

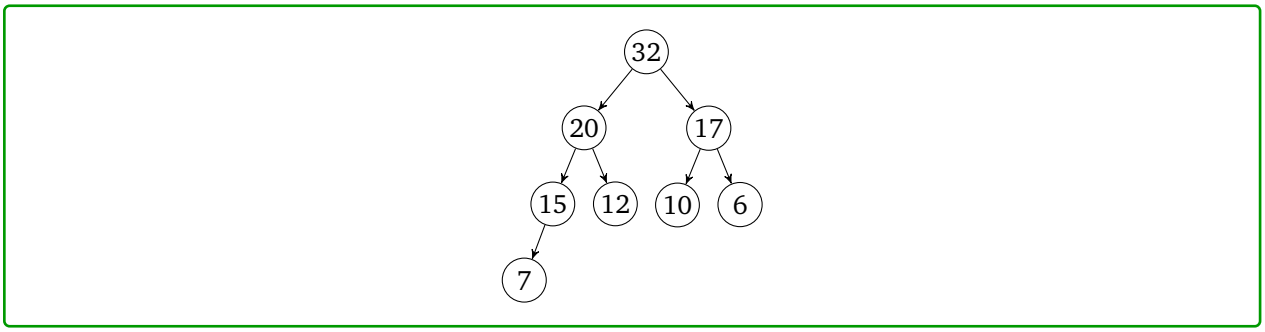
We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

3. Heaps

(a) Insert the following sequence of numbers into a *max heap*:

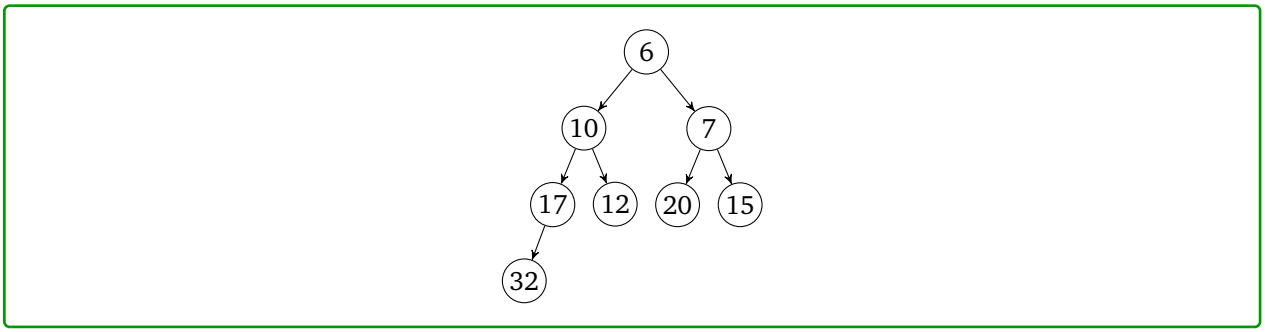
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



(b) Now, insert the same values into a *min heap*.

Solution:



4. Analyzing dictionaries

(a) What are the constraints on the data types you can store in a Red-Black tree? When is a Red-Black tree preferred over another dictionary implementation, such as a HashMap?

Solution:

Red-Black trees are similar to TreeMaps. They require that keys be orderable, though not necessarily hashable. The value type can be anything, just like any other dictionary. A perk over HashMaps is that with Red-Black trees, you can iterate over the keys in sorted order. Red-Black trees have a better worst case bounds for insert, delete, and remove, since they are always balanced; while hashtables can have $\mathcal{O}(n)$ operations in the worst case. When the hash function works well, though, hash tables are more efficient ($\mathcal{O}(1)$ operations).

(b) When is using a BST preferred over a Red-Black tree?

Solution:

One of Red-Black's advantages over BST is that it has an asymptotically efficient find() even in the worst-case.

However, if you know that find() won't be called frequently on the BST, or if you know the keys you receive are sufficiently random enough that the BST will stay balanced, you may prefer a BST since it would be easier to implement. Since we also don't need to worry about performing rotations and keeping track of red/black links, using a BST could be a constant factor faster compared to using a Red-Black tree.

5. Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or Red-Black tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or a Red-Black tree).

We can modify our second solution in a similar way by using specifically a BST or a Red-Black tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the Red-Black and BST tree's iterator will naturally print out the trains in the desired order.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either a Red-Black tree or a BST, since we can list out the trains in sorted order based on the ID.

Note that while the Red-Black tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n, since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 element is realistically going to be a fast operation.

What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.

Food For Thought

6. Heaps: Sorting and Reversing

(a) Suppose you have an array representation of a heap. Must the array be sorted? **Solution:**

No, $[1, 2, 5, 4, 3]$ is a valid min-heap, but it isn't sorted.

(b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap? **Solution:**

Yes! Every node appears in the array before its children, so the heap property is satisfied.

(c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap? **Solution:**

No. For example, $[1, 2, 4, 3]$ is a valid min-heap, but if reversed it won't be a valid max-heap, because the maximum element won't appear first.

7. Algorithm design

When writing mathematical expression, we typically write expressions in *infix* notation: in the form `NUM OPERATOR NUM`. An example of an expression written in infix notation is $4 + 6 * 5$. This expression evaluates to 34.

An alternative way we can write this expression is using *post-fix* notation: in the form `NUM NUM OPERATOR`. For example, consider the following expression written in post-fix notation:

$$4, 6, 5, *, +$$

This expression is interpreted in the following way:

- Read and store 4
- Read and store 6
- Read and store 5
- Multiply the last two stored values (and remove them from storage), then store the result
- Add the last two stored values (and remove them from storage), then store the result

The last result stored is the final "output". In this case, the expression above also evaluates to 34.

(a) Explain how you might apply or adapt the ADTs and data structures you've studied so far to evaluate an expression written in post-fix notation. Assume you accept the expression you need to evaluate as a string.

Solution:

Split the string (e.g. by doing `input.split(", ")` in Java) to get an array containing each number or operator.

Next, create a stack of ints, and iterate through the array.

Every time we encounter a number (or rather, anything we don't recognize as being an operator), convert that string to a number and push it onto the stack.

Every time we encounter an operator, pop the last two values on the stack and perform that operation. (We would likely need to hard-code how to handle each operator in a large if/else if/else branch or something). Push the result back on to the stack.

Once we finish handling every element in the array, pop whatever value we have left and return that.

There would obviously be some error checking embedded to avoid problems if the input isn't of valid format, but we typically don't include error checking in algorithm descriptions or pseudocode.

(b) Give pseudocode for this algorithm.

Solution:

```
int evaluateExpr(String expr)
    Stack s = new Stack()
    String[] exprs = expr.split(',')

    for (String s : exprs)
        if s == '+'
            int num1 = s.pop()
            int num2 = s.pop()
            s.push(num1 + num2)
        else if ... // other expressions not included, very similar to +
        else // no more exprs, probably a number
            s.push(parseInt(s))

    return s.pop()
```

Challenge Problems

8. Random Hash Functions

In class we talked about various strategies to minimize collisions. In this question we discuss how to use randomness to "spread out" collisions from a small set of very bad inputs into a larger set of almost-always-fine inputs. The last two parts of this problem are beyond the scope of this course, but are interesting nonetheless.

For simplicity, assume our keyspace (the set of possible keys) is the set $\{0, 1, 2, \ldots, 2^{30} - 1\}$. Suppose we have a hashtable of size 2^{10}. Let a be an odd integer less than 2^{30}.

Consider the hash function

$$h_a(x) = \left\lfloor \frac{(ax) \mod 2^{30}}{2^{20}} \right\rfloor$$

Notice that the function changes depending on the value of a we choose, so this is really a set of possible functions.

(a) Show that for any a, h_a outputs an integer between 0 and $2^{10} - 1$ (i.e. we can use this as a hash function for our table size)

Solution:

The numerator of the fraction is always a number from 0 to $2^{30} - 1$ (after we do the mod operation). Dividing by 2^{20} moves the number into range 0 to $2^{10} - 1/2^{20}$. When we take the floor, we round the numbers down to the next integer, so the range of possible outputs becomes 0 to $2^{10} - 1$, i.e. exactly the indices for a 0-indexed table of size 2^{10}.

(b) Choose $a = 1$, i.e. the hash function simplifies to

$$h_1(x) = \left\lfloor \frac{x \mod 2^{30}}{2^{20}} \right\rfloor$$

For this function, find a large set of elements that all hash to 0.

Solution:

The keys $\{0, 1, 2, \ldots, 2^{20} - 1\}$ all hash to 0 (modding by 2^{30} doesn't affect small values, and the floor rounds any number less than 1 to 0). For $a = 1$ this function hashes contiguous sets of 2^{20} numbers into each bin. For other hash functions, it is harder to find this set, but every hash function has this problem: if the key-space is much larger than the size of the table, there must be a large number of values that all collide.

(c) Let x, y be any of the two elements you found in the last part. Choose a few thousand values of a, and check whether $h_a(x) = h_a(y)$ for each of them (write code for this part). For what fraction of these hash functions do x, y collide? If the values of the hash function were totally random, how often would you expect collisions?

Solution:

The exact number of collisions you see will depend on which values of a, x, y you check, but you should see between 0.1% and 0.2% of hash functions causing a collision.

If the outputs of the hash function were truly random, we would see a collision with probability equal to $\frac{1}{\text{table size}}$, i.e. $1/1024$ or about $.1\%$ of the time. So the output we're seeing as we change a is *nearly* as good as really random outputs.

(d) The following statement is true (explaining why is beyond the scope of the course): For any x, y if you choose a at random, the probability that $h_a(x) = h_a(y)$ is at most $2/2^{10}$.

Use this fact, or your observations in the last part, to explain why you might decide to choose a random a instead of just choosing $a = 1$ (hint: imagine you know someone is using the hash function with $a = 1$, how can you use the first part to slow their code down? Can you do the same for a random a?)

Solution:

A user will have a fixed set of keys they need to use. If these happen to be keys that all hash to the same place, the hash table will have very poor performance, and the user has no hope of fixing this (because they can't really change their keys). On the other hand, if we choose a random hash function, with high probability (say 99.9%) the fixed set of keys won't be a problem (though there is still a small chance of getting the poor performance).

Occasionally, we worry about an attacker intentionally giving us bad data to try to break our code. If an attacker knows your hash function, they can do what we did in the first part, and find a set of inputs that will slow down your hash table. On the other hand, if you're choosing a hash function randomly, there is no single set of inputs that can cause bad performance. Choosing a function randomly lets us "spread out" the bad behavior across inputs. Said a different way:

- With a single, fixed function for most inputs, things work great; but there are some very bad inputs on which things work terribly.
- If you choose a random hash function, every input has a very high probability of good performance, but on many inputs there is a very small chance of bad performance.

There are a lot of mathematical caveats here (e.g. you need a good set of hash functions to choose from, your choice of hash function needs to be random enough that it can't be predicted, etc.) but we don't have time to go into them. See http://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-hashing.pdf for more information on this hashing scheme, and randomized hashing in general, including the formal mathematics.